UNCLASSIFIED

DEFENCE RESEARCH ESTABLISHMENT
CENTRE DE RECHERCHES POUR LA DÉFENSE
VALCARTIER

DREV - TR - 1999 - 145
Unlimited Distribution / Distribution illimitée

# DEBUGGING AND ANALYSING WINDOWS NT PROGRAMS AT THE BINARY LEVEL

by

Martin Salois

May/mai 2001

Approved by / approuvé par

Section Head / Chef de section

19 mai 01

Date

SANS CLASSIFICATION

102-01710/

**WARNING NOTICE**

The information contained herein is proprietary to Her Majesty and is provided to the recipient on the understanding that it will be used for information and evaluation purposes only. Any commercial use, including use for manufacture, is prohibited. Release to third parties of this publication or of information contained herein is prohibited without the prior written consent of DND Canada.

© Her Majesty the Queen in Right of Canada as represented by the Minister of National Defence, 2001

## ABSTRACT

In recent years, civilian and military organisations have been using Commercial-Off-The-Shelf (COTS) software more and more instead of developing their own software. Even though this approach offers many advantages in terms of development delays and costs, it causes great concern when it comes to security assurance since it is much more difficult to test software when the source code is not available, as is the case for most COTS software.

In 1997, a project called MaliCOTS was initiated to investigate ways of detecting and eliminating the threat caused by malicious codes that could be present in COTS programs. This report represents a first step toward overcoming the technical difficulties associated with a dynamic approach in which one directly monitors the execution of the program.

To monitor a program's execution at the binary level, basic knowledge of assembly language programming is required. It is also preferable to have a good understanding of the architecture of the operating system in use — in this case Windows NT running on an Intel machine. Finally, to be able to control a program's execution and look at its private data, it is necessary to know how to implement a debugger.

This report presents information that was gathered from numerous heterogeneous sources and experimentation, thus allowing a programmer to implement and use a simple debugger that works at the binary level in Windows NT.

## RÉSUMÉ

Depuis quelques années, les organisations civiles et militaires utilisent de plus en plus les logiciels du commerce (COTS) plutôt que de développer des programmes originaux. Même si cette approche offre son lot d'avantages en terme de coût et de temps de développement, elle cause de graves soucis en ce qui a trait à l'assurance sécurité. Il est beaucoup plus difficile de tester un programme lorsque nous n'avons pas accès au code source, ce qui est le cas pour la plupart des logiciels commerciaux.

En 1997, un projet, nommé MaliCOTS, a débuté pour trouver des façons de détecter et d'éliminer la menace causée par du code malicieux qui pourrait être présent dans les logiciels commerciaux. Ce rapport présente un point de départ pour surmonter les difficultés techniques associées à une approche dynamique dans laquelle l'exécution du programme est surveillée très étroitement.

Pour examiner l'exécution d'un programme au niveau binaire, une connaissance de base du langage de programmation assembleur est requise. Il est également préférable d'avoir une bonne compréhension du système d'exploitation utilisé (dans ce cas, Windows NT s'exécutant sur une machine Intel). Finalement, pour être à même de contrôler l'exécution du programme et d'examiner ses données privées, il est nécessaire de savoir comment implanter un dévermineur (debugger).

Ce rapport présente toute l'information regroupée à partir de sources hétérogènes et d'expérimentations afin de permettre à un programmeur d'implanter et d'utiliser un simple dévermineur fonctionnant au niveau binaire dans Windows NT.

## TABLE OF CONTENTS

ABSTRACT / RÉSUMÉ . . . . . i
EXECUTIVE SUMMARY . . . . . iii
LIST OF ACRONYMS . . . . . iv
LIST OF FIGURES . . . . . vi
LIST OF TABLES . . . . . vii
LIST OF CODE EXCERPTS . . . . . viii

**1.0 INTRODUCTION** . . . . . **1**

**2.0 WINDOWS NT SECURITY FEATURES** . . . . . **3**
2.1 Overall System Architecture . . . . . 3
2.2 Windows NT Security . . . . . 10

**3.0 WIN32'S DEBUGGING FEATURES** . . . . . **17**
3.1 Basic Assembly Required . . . . . 17
3.2 Intel Processor's Architecture for Debugging . . . . . 24
3.3 Protected-Mode Memory . . . . . 28
3.4 Inside Win32's Debugging API . . . . . 31

**4.0 CONCLUSION** . . . . . **41**

**5.0 REFERENCES** . . . . . **43**

INDEX . . . . . 45
FIGURES 1 TO 10

## EXECUTIVE SUMMARY

Military information systems are built more and more with Commercial-Off-The-Shelf (COTS) software. Since most of these programs or libraries are proprietary, one has limited control over quality and security. The source code is usually not accessible. This can severely limit the number of tests that can be performed on a product and thus its degree of reliability. Obviously, this is unacceptable in most military situations.

To address this problem, a project was initiated in 1997 for a collaboration between the Interoperability Group at Defence Research Establishment, Valcartier (DREV) and the Laval University's Languages, Semantics and Formal Methods (LSFM) Group. As a result, many research avenues were defined and are currently being investigated.

Among these avenues, there is dynamic analysis. In this approach, the runtime behaviour of a program is examined. There are two main ways to do this. The first is to examine second-hand information, such as system logs and resource access control. The second way is to directly control the flow of execution and examine the code at the binary level.

There are several difficulties associated with this last technique. First, it requires a knowledge of assembly language programming that today's programmers do not generally possess. Next, it requires an intimate knowledge of the operating system's inner workings -- in this case Windows NT. Last, but not least, a way to control the flow of execution and to look at the data of the program is required. In a sense, a form of advanced debugger is needed.

While performing research toward building such a debugger, it was found that Win32, the basic interface to Windows NT's functionalities, provides all the necessary debugging functions. Unfortunately, the documentation on the subject is somewhat lacking, if not totally absent.

The aim of this report is to present all the basic information necessary to understand and implement a basic debugger. The information, gathered from various sources, and the experience gained by the author is presented in an organised fashion. It is hoped that it will provide a great time saver for anyone interested in the subjects presented. In addition to describing how to build a debugger, the report gives a crash course on Windows NT's architecture, its security features, and the assembly language for an 80x86 platform. This provides the necessary a priori knowledge needed to understand the debugger.

This preliminary work is the first step toward a dynamic analysis tool that could be developed to test, verify, and validate commercial programs before their inclusion in operational command and control information systems. These verifications are, of course, a necessary step before widespread use of an unknown program. In the future, if computing performance is sufficient, such a tool could also be used for constant protection, much like current virus detection tools.

## LIST OF ACRONYMS

**ACL** Access Control Lists

**AF** Adjust Flag

**API** Application Programming Interface

**CF** Carry Flag

**CPU** Central Processing Unit

**COTS** Commercial-Off-The-Shelf

**DLL** Dynamic Link Library

**DMA** Direct Memory Access

**DREV** Defence Research Establishment, Valcartier

**FSD** File-System Driver

**GB** Gigabyte

**GDI** Graphical Device Interface

**GDT** Global Descriptor Table

**GUI** Graphical User Interface

**HAL** Hardware Abstraction Layer

**ICSA** International Computer Security Association

**IF** Interrupt Enable Flag

**I/O** Input/Output

**LAN** Local Area Network

**LDT** Local Descriptor Table

**LIFO** Last In First Out

**LOFO** Last On First Out

**LPC** Local Procedure Call

**LSFM** Languages, Semantics and Formal Methods

**MaliCOTS** The name of our project: MALIcious code detection in COTS software

**MMCC** Malicious Mobile Code Consortium

**NCSC** National Computer Security Center

**NIST** National Institute of Standards and Technology

**NTFS** NT File System

**OF** Overflow Flag

**PF** Parity Flag

**RAS** Remote Access Service

**RPC** Remote Procedure Call

**RF** Resume Flag

**SF** Sign Flag

**TF** Trap Flag

**TCSEC** Trusted Computer System Evaluation Criteria

**VDM** Virtual DOS Machine

**WOW** Windows on Windows

**ZF** Zero Flag

## LIST OF FIGURES

1 Overall architecture of the NT kernel mode components . . . . . 4
2 Detailed view of the Executive . . . . . 5
3 The Win32 subsystem . . . . . 9
4 The 80x86's general purpose registers . . . . . 19
5 The stack . . . . . 21
6 The debug registers . . . . . 25
7 EFlags Register . . . . . 27
8 Protected-mode address translation process . . . . . 29
9 Virtual to linear address translation . . . . . 30
10 Components of a linear address . . . . . 31

## LIST OF TABLES

I Essential conditional jump instructions. . . . . 23

P515758.PDF [Page: 14 of 63]



## LIST OF CODE EXCERPTS

1 The CONTEXT structure . . . . . 35
2 The DEBUG_EVENT structure . . . . . 36
3 The EXCEPTION_DEBUG_INFO and EXCEPTION_RECORD structures . . . . . 37
4 The main debugging loop . . . . . 39
5 Putting in a breakpoint (int 3) . . . . . 40

## 1.0 INTRODUCTION

Competition between enterprises is fiercer than ever. To stay in the race, companies must keep productivity at its peak and cost as low as possible. One of the solutions that seems to prevail is to increase use of the so-called information technologies as they become available.

The problem with this approach is that the development of software programs requires a high level of competence, skills, and knowledge. This, in turn, demands highly paid personnel, high maintenance, and extensive verification of the software.

For these reasons, companies enthusiastically welcome ready-to-run applications that promise no end to their usefulness. Such applications are called Commercial-Off-The-Shelf (COTS) software. COTS software includes development tools and programming libraries (Visual C++, Delphi), office processing tools (MS Office, COREL suite), networking software (Windows NT, Netware), and so on.

Unfortunately, there is at least one problem when it comes to using software that was developed commercially: you do not know *what* has been developed and *how* it was developed. This can cause severe security and safety risks because some vendors will take any shortcuts possible to reduce "time to market". Because of this, it is more than likely that errors remain in the software.

Furthermore, it is a known practice for companies to insert some protection schemes in their software to prevent illegal copies from being made, or to prevent using a trial version after the expiration date. Arguably, protecting their software may not be perceived as a "malicious" action on the part of the companies involved. The problem is not the protection itself, but rather how it is implemented. The schemes used to protect software generally involve going around the system's normal functionalities, and this can clearly become malicious. There are also reported cases of trapdoors inserted into programs that give access to personal confidential data or to secret services, or to anyone able to open the trapdoor.

These pieces of code that trigger and/or perform an unwanted action in a program, or a system, notwithstanding the seriousness of this action, are called *malicious codes*. Potential vulnerabilities that could be used with malicious intent are also included in this definition.

In 1997, a project titled MaliCOTS was initiated to investigate ways of discovering and eliminating the threat of malicious code in COTS software — the platform of choice being Windows NT, running on an Intel processor. The project currently involves two DREV scientists, four professors from the Languages, Semantics and Formal Methods (LSFM) Group at Laval University, and about twelve of their graduate students.

Two main classes of tools that could be used to achieve this goal are static analysis tools and dynamic analysis tools. In static analysis, the program code is examined— be it source code or disassembled code — to try to predict the behaviour that it will have at runtime. In dynamic

analysis, the actual runtime behaviour is investigated.

There are two ways of examining the runtime behaviour of a program: passive and active monitoring. The former technique examines second-hand data such as input/output and logs; the latter directly influences the program under study. A debugger is a good example of active monitoring. Active monitoring was examined more closely initially since it can possibly prevent a malicious action from being executed. Passive monitoring, by definition, can only discover the malice after the fact and, thus, cannot prevent its execution.

Noting that an active monitoring tool closely resembles a debugger, the first step in our work was to find out if one could customise a commercial debugger. There are many very good debuggers on the market, but none could be customised for the MaliCOTS project special needs. Thus, it was decided to investigate the development of a debugger to fulfil these needs.

It turns out that once one undertands how, developing a debugger is a relatively simple task. Unfortunately, knowing how is a big problem since the information about low-level programming in Windows NT is scarce. Closely examining the documentation of Microsoft Visual C++ (Ref. 1) was not much help because it documents the debugging functions of Win32, but not its structures and not how to use them. Intel's documentation (Refs. 2, 3, 4) was better but not quite enough. After that, the project personnel looked for books on the subject and did not find any. Finally, attention was turned to "hackers", who have acquired a great deal of experience on the subject by cracking and generally experimenting with the limits of operating systems. An informal but almost complete article describing many NT debugging structures and written by the colourfully named Iceman (Ref.5) was discovered.

After a great deal of experimentation, sufficient expertise and information was acquired. MaliCOTS project personnel could indeed develop a debugger in Windows NT if needed.

In order to save an enormous amount of time for future researchers on the MaliCOTS project and for anyone else requiring information on this subject, this report presents all the ingredients needed to get started at building and using an NT debugger. First, Windows NT's architecture is presented, with emphasis on its security features (Chapter 2). Then, the debugging features of Win32, the main subsystem of NT, are presented along with how to use them to implement a debugger (Chapter 3). Conclusions are presented in Chapter 4.

This work was carried out between August and December 1998 under TNS 5BF19, "MaliCOTS".

# 2.0 WINDOWS NT SECURITY FEATURES

Before discussing the security features of Windows NT, it is important to obtain a general understanding of how Windows NT works. For this reason, this chapter begins with an overview of Windows NT's architecture. The following section surveys the security features. The purpose of this chapter is to obtain a working knowledge of Windows NT and the terms that it uses, in order to have a better grasp of the background environment of the malicious code project.

## 2.1 Overall System Architecture

This section is largely inspired by Chapter 1 of Art Baker's book (Ref. 6). All the figures in this section are reproduced from there. It presents the main design decisions that underlie the actual Windows NT operating system, without going too deep in the details. For more details, refer to Ref. 6.

### 2.1.1 Design Goals

The development team of Windows NT had the following design goals in mind:

- *Compatibility.* Support for existing software and legacy hardware.
- *Robustness and reliability.* Resistance to naïve or malicious users, and isolation of applications — meaning that an application runs in its own protected address space without, in theory, being able to influence other applications.
- *Portability.* Support for current and future hardware platforms.
- *Extendability.* Support for adding new features and new hardware.
- *Performance.* Reasonable performance on available hardware and support for multiprocessor architectures.

Balancing all these goals is a complex process and the following subsections describe the solution that the Windows NT development team proposed.

### 2.1.2 Hardware Privilege Levels

Applications in a multitasking environment should not be allowed unlimited access to all computing resources. For example, they should not be allowed to manipulate values in the memory space of other processes. Since it is doubtful that all applications will respect this, hardware that

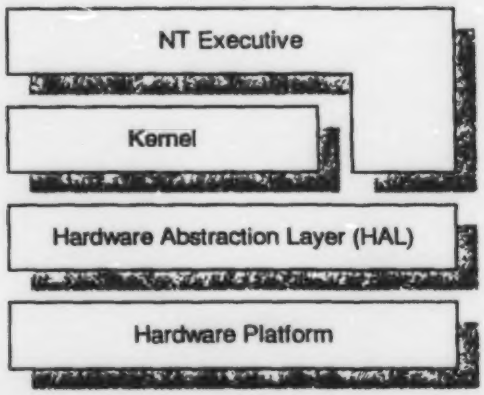


FIGURE 1: Overall architecture of the NT kernel mode components

wants to run Windows NT must be able to provide two Central Processing Unit (CPU) execution modes:

- *Kernel mode.* When the CPU runs in kernel mode, it has every possible privilege.
- *User mode.* When the CPU runs in user mode, it prevents access to privileged instructions, and checks memory references to make sure that they belong to the process. The operating system is in charge of the gate that allows access to the kernel mode.

### 2.1.3 Base Operating System Components

As seen in Fig. 1, three major blocks implement the general operating system.

#### 2.1.3.1 Hardware Abstraction Layer (HAL)

The HAL is a thin layer of software routines that allows the rest of the system to have a uniform interface to any hardware, apart from the CPU, by hiding platform-specific details. It provides functions for:

- Off-chip caches
- Timers
- Input/Output (I/O) buses
- Device registers

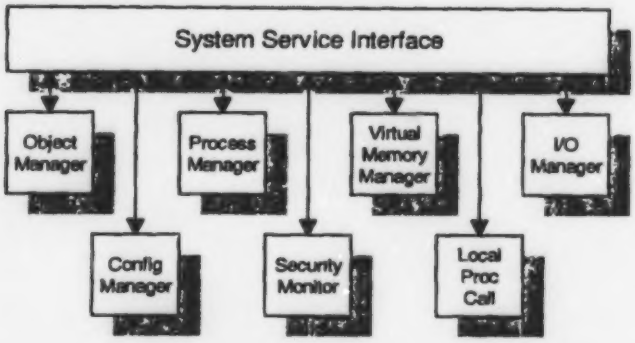


FIGURE 2: Detailed view of the Executive

- Interrupt controllers
- Direct Memory Access (DMA) controllers

#### 2.1.3.2 Kernel

The kernel presents a simplified view of the CPU itself. It handles:

- Interrupt and exception dispatching
- Thread scheduling and synchronisation
- Multiprocessor synchronisation
- Time keeping

With the help of these services, the upper layers can basically ignore the architecture of the underlying CPU. An interesting feature is that they are presented as either dispatcher objects or control objects. Dispatcher objects handle primarily the management and synchronization of threads, while control objects affect the behaviour of the operating system in some way.

#### 2.1.3.3 NT Executive

This is the key element of the Windows NT operating system. The Executive uses the HAL and the Kernel to provide all the typical functions that are required by an operating system. This makes it portable across different CPU architectures.

As can be seen in Fig. 2, the Executive is composed of different software components that offer their services to user mode components and to each other. The components are modular and independent. Brief descriptions of each of them follow.

**System Service Interface** It is often necessary for normal applications to execute some kernel mode code — for example, to access files, or to access hardware. In Windows NT, the system service dispatcher (part of the interface) uses the CPU's exception mechanism to provide services that are required to be run in kernel mode.

**Object Manager** Like the Kernel, the Executive provides its services through an object-based interface. The objects represent entities such as files, processes, threads, and shared memory segments. The Object Manager is in charge of managing these objects. This allows for easier tracking of resources and enforcement of security.

**Configuration Manager** The Configuration Manager is in charge of keeping track of all hardware configurations. It does this with the aid of the Registry. For instance, the Registry is used to:

- Identify device drivers as trusted components.
- Find and allocate peripheral hardware.
- Set up error-logging message files.
- Enable driver-performance measurement.
- Store each user's preferences.
- Store applications' configurations and install/uninstall options.

The Registry works by storing keys and their values for the different parameters.

**Process Manager** A process is the entity level at which resource tracking and security access checking is performed in Windows NT. A process holds resources and a security identity, and it has its own virtual address space. A process also contains one or more threads. A thread is the entity that receives CPU time to execute.

The Process Manager is in charge of creating, managing, and deleting processes and threads. Mostly, it is an elaborate interface to the relevant Kernel mechanisms.

**Security Reference Monitor** This Executive component's task is to enforce the system's security policy; it does not define the security policies, it just enforces them. Among other things, it provides services to validate accesses to objects, checks for user privileges, and generates audit messages.

**Virtual Memory Manager** Under Windows NT, each process has a flat 4-gigabyte virtual address space. The lower half of this space contains process-private code and data along with the process's stack and heap space. It also contains any DLLs (Dynamic Link Library) and File Mapping object the process may need. The upper half contains only kernel mode code. The main job of the Virtual Memory Manager is to give the illusion that a process has access to this huge address space by handling paging mechanisms.

**Local Procedure Call (LPC) Facility** The LPC facility is a message-passing mechanism used for communication between processes on the same machine.

**I/O Manager** Through a consistent interface, the I/O Manager transforms I/O requests into properly sequenced calls to various device driver routines.

### 2.1.4 Extensions to the Base Operating System

The Executive components provide the general services of an operating system. They are not meant to be directly used by upper layers of software; they are not even publicly documented. Users and programmers of Windows NT are supposed to deal with *protected subsystems*.

Depending on the work they perform, protected subsystems are divided in two major categories:

- *Integral subsystems.* An integral subsystem performs necessary operating system functions. Here are a few examples:
  - Together with the Security Accounts Manager and the Logon process, the Local Security Authority defines security policies for the system.
  - The Service Control Manager loads, supervises, and unloads trusted system components like services and drivers.
  - The Remote Procedure Call (RPC) Locator and RPC Service processes give support to distributed applications that use the RPC service.

- *Environment subsystems.* These are protected subsystems, whose job is to provide a programming interface and execution environment for application programs native to some specific operating system. Currently, Windows NT provides the following subsystems:
  - *Win32.* The 32-bit native mode programming interface for Windows NT and Windows 95/98.
  - *Virtual DOS Machine (VDM).* The VDM allows Windows NT to run 16-bit MS-DOS applications.
  - *Windows on Windows (WOW).* With the help of the VDM, WOW allows Windows NT to run 16-bit Windows applications.
  - *POSIX.* This subsystem provides compatibility with the POSIX 1003.1 source-code standard.
  - *OS/2.* This subsystem creates an execution environment for IBM's 16-bit OS/2 applications (available only on 80x86 versions of Windows NT).

Win32 is the main subsystem of Windows NT and all current applications are developed on it. As such, more details are warranted and are presented in the next subsection. The other subsystems are provided mainly for compatibility and may suffer from performance degradation. For example, in the VDM, although it may seem as if one has direct access to hardware via interrupts, those interrupts are actually simulated by the VDM.

#### 2.1.4.1 More about the Win32 Subsystem

The Win32 subsystem is the owner of the screen, the keyboard, and the mouse. As such, it manages all graphical I/O for the whole system, including the other subsystems. It also provides the Win32 Application Programming Interface (API), which is used by applications to communicate with the Executive.

Because of its special status as the main subsystem, Win32 is implemented a little differently from the others. Figure 3 shows the organisation of the Win32 subsystem.

For performance reasons, the Win32 subsystem does not work entirely in user mode. Some of it is implemented in kernel mode. You can divide its functions into three categories:

- The USER functions manage Graphical User Interface (GUI) objects like menus and buttons.
- The Graphical Device Interface (GDI) functions perform low-level drawing operations on graphical devices like the screen, or printers.

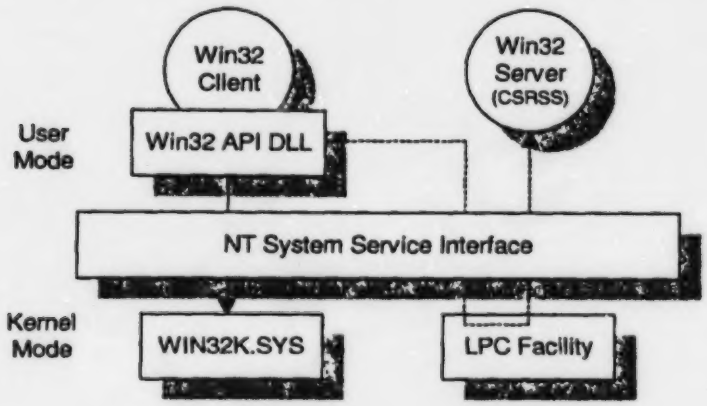


FIGURE 3: The Win32 subsystem

- The KERNEL functions manage things like processes, threads, synchronisation objects, shared memory, and files. These functions match very closely the services provided by the Executive.

Originally, in previous versions of Windows NT, the Win32 Server handled all GUI policy-making code. Again, for performance reasons, most of these functions have been moved to the `Win32K.sys` component, which runs in kernel mode.

In effect, the Win32 subsystem is the operating system from the user's and most developers' points of view. Almost exclusively, developers that work at the hardware level need to be concerned with lower layers of the system, such as the Executive. The Win32 subsystem provides APIs for file access, dialog display, network communications, graphical display, sound playing, and so on. As Chapter 3 shows, it also has major debugging facilities and it provides almost direct access to the processor's functionalities.

#### 2.1.4.2 File-System Driver (FSD)

FSDs are responsible for keeping the on-disk structure required by various file systems. Microsoft currently supports the following FSDs:

- *FAT.* File-Access Table: Windows 95/98 extended MS-DOS file system.
- *NTFS.* NT File System (NTFS): Windows NT high reliability file system.
- *HPFS.* OS/2 high performance file system.

- *CDFS.* ISO 9660 CD-ROM file system.
- *MSFS.* Mailslot file system.
- *NPFS.* Named pipe file system.
- *RDR.* Local Area Network (LAN) Manager redirector.

NTFS is the norm when it comes to installing a *secure* Windows NT system. It provides support for Access Control Lists (ACL), one of the main features of Windows NT security, as we will see in the next section.

## 2.2 Windows NT Security

This section is largely inspired by the book of Stephen A. Sutton (Ref. 7). It gives a working knowledge of the terms used by Windows NT security features.

### 2.2.1 Accounts and Authentication (Logon)

The user account is the fundamental concept behind Windows NT security. Through the account and its password, one must identify oneself to Windows NT before one can use it. This process is generally called "authentication" or in Windows NT, more simply "logon".

All rights and privileges come from one's account. Consequently, only users with certain administrator privileges can create and modify accounts. Accounts can be valid for a single computer or for various groups of computers, which are called domains. When one requests access to resources on another computer, that computer transparently authenticates from the current session's account and password.

The use of accounts allows multiple users to share a single computer. The preferences of each user are kept and can be modified independently of each other. Administrators can also control exactly which applications any user has the right to execute and they can reserve some directories to the exclusive use of one user (of course, administrators could access them by force if the need arises).

Windows NT also provides various logon and password checking and restrictions. Administrators can impose a minimum password length, a minimum and a maximum password age, and a password uniqueness policy that the administrator can set to record the $x$ previous passwords. This permits the administrators to ensure that a user does not cycle through a list of a few passwords that are changed every so often, prevents using no password at all, or using an unsafe one. There is also an account lockout feature. The administrator can set it so that it will prevent access to the account for a fixed amount of time if $x$ logon attempts fail in a certain period of time.

### 2.2.2 Domains

Administrating a network comprising a large number of workstations would be quite complicated if the administrator had to synchronise the accounts and security policies of each workstation separately. Domains are the logical solution to solve this problem. A *domain* is an enclave that regroups several workstations that logically have the same use. Workgroups can be formed this way, and every computer in one workgroup can have the same settings; these settings only need to be specified once.

Hence, a user typically has an account in a domain. It gives access to all computers in the domain. Of course, the administrator can also give that user special privileges on some specific computers. For example, it makes sense for users to have complete control over their own computers, although some administrators will prefer to keep complete control for themselves.

A domain $D_A$ can trust another domain $D_B$. This means that users from $D_B$ can log on to $D_A$. The reverse is not necessarily true. Just because $D_A$ trusts $D_B$ does not mean that $D_B$ trusts $D_A$. This means that one-sided relationships are created. If one wants $D_B$ to also trust $D_A$ then one has to initiate the trust procedure a second time the other way around.

Finally, administrative rights can scale from a single workstation to multiple domains. For instance, users can be all-powerful on their own workstation and not have any power on others, while an administrator can be all-powerful on some workstations, all the workstations of a single domain, all the workstations of some domains, or every single computer in a company's network.

### 2.2.3 User Groups

Any user can create groups of users to be used as a set containing many users. The advantage is that one entity, the group, is used in places where one wants to bestow the same rights to more than one person. For example, one could create a group called `Friends` to identify close friends and give them all the same kind of access to one's computer. Later on, new friends can just be added to the group and they automatically get access to all the resources that the group has access to. It is a great way to minimize the amount of maintenance needed on the access control lists.

### 2.2.4 Access Control Lists (ACL)

Together with the account and password mechanism, Access Control Lists (ACL) constitute the main part of Windows NT security. As seen earlier, the NTFS disk format is needed to use ACLs.

Each file and each directory has an ACL as do shared resources like printers and network directories. An ACL controls exactly which users can access a resource and their set of privileges

by maintaining a list of users or groups.

The ACLs of each and every single file do not need to be considered. A mechanism exists that allows the user to set the default ACL of new files created or imported into a directory. First, define a logical directory tree. Then, all that is needed is to set the ACL of special files that need more (or less) protection. For files, the following are the possible ACL privileges:

- Read (R)
- Write (W)
- Execute (X)
- Delete (D)
- Change Permissions (P)
- Take Ownership (O)

The first four are self-explanatory. "Change Permissions" allows the user to change the privileges' setting on the file. "Take ownership" allows the user to become the owner of the file. Since the owner of a file always has full control over it and since the P privilege allows users to give themselves the O privilege, giving either one of them has the same effect as giving them both. It is more a question of semantics and general good use than a question of effective usefulness.

For directories and printers, there are more privileges that need to be taken into account. For directories, one needs to specify the access of newly created files. In this case, a special user name exists, called `CREATOR OWNER`. It permits the specification of a special set of privileges for the user who creates the file. With this one can, for example, use a common directory, like temp, and restrict access to only the file creator. In the case of printers, one can specify print privileges, document management, or full control.

The granularity of the ACLs mechanism is quite impressive. One can specify access restrictions down to every single user. When a conflict arises, a pessimistic approach prevails. For example, if user `Toto` is in the group `Friends` but one decides that he is not a close enough friend to have access to personal pictures, one can still give picture viewing privileges to the group `Friends` but specify that user `Toto` has no access.

### 2.2.5 Trusted Path

The Trusted Path is the first line of defense of Windows NT. Through it, one can log in, log out, change its password, lock the workstation, and access the task manager.

The Trusted Path is a special sequence of keys, `CTRL-ALT-DEL`, that one presses simultaneously. It gives access to a special window entitled "Windows NT Security". Windows NT guarantees that this window, when accessed via the Trusted Path, is legitimate and trustworthy. In theory, there is no way to bypass the feature.

Users should always use the Trusted Path when dealing with passwords since it is the only way to be sure that the program asking for a password is legitimate and not a malicious one, masquerading as the real thing. The author has tested the Trusted Path against keylogging programs, and it is worth noting that they do not work. Everything typed outside of the Trusted Path is recorded by the keylogger. Were one to enter a password with the keylogger activated one would, in effect, reveal this password.

### 2.2.6 Flexible, Role-Based Administration

When it comes to defining exactly what a user can and cannot do, Windows NT benefits again from a good level of granularity.

While it makes sense to have only a very few all-powerful administrators, this creates the problem that a few individuals cannot perform the very large number of task often associated with the management of a medium or large network. Windows NT comes to the rescue by providing a way to create lesser administrators that have only specific privileges. The following special groups are created by default:

- *Administrators.* Members can fully administer the computer/domain.
- *Backup Operators.* Members can bypass file security to back up files.
- *Power Users.* Members can share directories and printers and be all-powerful on a single workstation that does not concern security. They act as mini-administrators of their own personal workstation.
- *Replicators.* Support file replication in a domain.

Furthermore, if none of the built-in groups correspond exactly to one's needs, one can specify directly the kind of rights required for a certain user or group. The following rights exist:

- Access this computer from network.
- Back up files and directories.
- Change the system time.

- Force shutdown from a remote system.
- Load and unload device drivers.
- Log on locally.
- Manage auditing and security logs.
- Restore files and directories.
- Shut down the system.
- Take ownership of files or other objects.

With all of this, one can customise groupings and user policies according to one's own specific needs.

### 2.2.7 Security Auditing

By default, Windows NT logs its normal system maintenance. In addition, one can also decide to log security-relevant events. Once more, it has an incredible level of granularity because not only can one decide to audit file accesses, one can also set the auditing policy separately for each and every single file and directory, and for each and every single user. This may seem like too much information to deal with, but the job is made easier by the ability to filter events in the Event Viewer program. Also, one can turn the whole security auditing mechanism on and off.

As a whole, one can audit the success, failure, or both, of the following events:

- Logon and Logoff.
- File and Object Accesses.
- Use of User Rights.
- User and Group Management.
- Security Policy Changes.
- Restart, Shutdown, and System events.
- Process Tracking.

As to files and directories, one can set auditing to record the use of the six ACL privileges identified in Subsection 2.2.4 for success, failure, or both.

An auditing event record contains a lot of information that tells what happened, when it happened, who caused it, and so on.

### 2.2.8 Service-Specific Features

Some specific services have their own special security features. These are designed not to interfere with the normal Windows NT security features.

An example of such a service is the Remote Access Service (RAS). The RAS is optional in Windows NT. It allows a user to remotely connect on a workstation in a LAN, from a modem or the Internet, and to then act as if the user was in the LAN itself. The user can connect to shared resources on the workstation's domain. However, special restrictions may be applied to this type of connection.

### 2.2.9 The C2 Rating

There has been much talk about Windows NT and its C2 rating. Many things have been written that were not true, or only partially true. Therefore, a few words about it are justified to clarify.

In 1983, the U.S. government established an evaluation process for operating systems that remains mostly unchanged today. Two agencies cooperated to perform the security analysis: the National Computer Security Center (NCSC), and the National Institute of Standards and Technology (NIST). The published criteria are in a book called the *Trusted Computer System Evaluation Criteria (TCSEC)*. Because of the color of its cover, it is generally called the "Orange Book". The Orange Book only applies to an operating system in stand-alone mode, without network components. Separate criteria, which appear in the *Trusted Network Interpretation* (the Red Book) handles the situation with network components .

The Orange Book rating uses a progressive scale:

(*least secure*) **C1** → **C2** → **B1** → **B2** → **B3** → **A1** (*most secure*)

The problem with this evaluation is that an operating system needs all the features of a level to obtain a rating. It does not acknowledge the fact that one can have many, or event most, of the required features of a higher level. In the case of Windows NT, it has, according to Sutton (Ref. 7), features up to the B3 level. Unfortunately, it lacks some functionalities of the B1 level, so it is rated at the C2 level and obtains no recognition for its other features.

Finally, it is worth noting that only version 3.5 of Windows NT, with Service Pack 3, has been formally rated C2 (Ref. 8). It does not say anything about the current version of Windows NT (Version 4.0). In practice, for the C2 rating to apply, it means that not only must one use an older version of the operating system, but one must also comply with the given configuration guidelines. Among other things, it means that one has to (Ref. 9):

- Turn off networking completely (since NT is just evaluated on the Orange Book, not the Red).
- Disable floppy disk access.
- Change the standard file system permissions to be more restrictive.
- Change a lot of permissions in the registry.

### 2.2.10 Third-Party Tools

Because of its design, Windows NT makes it easy for third-party vendors to provide additional security services. Here are some of the more common ones that can provide a more *secure* environment:

- *Network Encryption.* Strangely enough, Windows NT does not provide a mechanism to encrypt communications over the network. A sniffer can catch a lot of illicit information.
- *Audit Trail Analysers.* A way to automatically analyse the records of the auditing mechanisms for anomalies.
- *Personal Identification Devices (PID).* Instead of identifying users only with something they "know", like the password, one can use something they "hold", like a card, a key, fingerprints, and so on.
- *Security-Smart Applications.* It is futile to have ACLs, secure passwords, a Trusted Path, and the like if an application bypasses all those features, for example, by saving protected information in an unprotected zone. Therefore, it would help a great deal if applications collaborated with the security features. For instance, they can provide useful audit events about their own activities.
- *Virus Checkers.* Using an up-to-date virus detection tool is paramount in maintaining the integrity of a system.

Of course, it would be nice also to have a more general malicious code detection tool.

# 3.0 WIN32'S DEBUGGING FEATURES

The MaliCOTS project requires the ability to perform some dynamic analysis at the assembly level. Since the work is on Windows NT (on an Intel platform), this requires knowledge of the inner workings of Win32. A debugger-like tool is required to be able to execute a program instruction by instruction.

Fortunately, Win32 provides all the debugging facilities that are needed. Unfortunately, these facilities are poorly documented to say the least. Bits of information can be found here and there in the Microsoft documentation, the Internet, and the Intel manuals, but it was impossible to find one coherent document that explained the process of creating a debugger in useful and practical detail.

In order to help future workers on the project, this chapter regroups the information that was gathered from these different sources and, more importantly, from a great deal of experimentation.

For a brief description of Win32 and to see where it fits into the Windows NT architecture, see Chapter 2.

## 3.1 Basic Assembly Required

Before constructing a debugger, a basic knowledge of 80x86 assembly language programming is required. Detailed knowledge and/or significant programming experience is not needed. What is needed is to be able to recognise simple and usual commands, and to understand the basic workings of the processor at this level. One needs to know what registers there are, what a stack is, and so on.

Gavin Estey (Ref. 10) has written a very good introduction to Intel's 80x86 assembly language, the chosen platform for the project. Information from that introduction has been summarised, with complementary and updated information included where necessary.

### 3.1.1 Introduction to the Hardware

In this subsection, the basic terminology of assembly language programming is presented.

#### 3.1.1.1 Representation of Numbers in Binary

As the reader probably already knows, numbers in computers are represented in binary (base two). Different terms are used to represent numbers of different sizes. If each *b* can be either a 0 or a 1, here are the essential ones:

- 1 BIT: *b*
  One bit is the simplest form of data. It represents either a 1 or a 0.

- 1 NIBBLE: *bbbb*
  4 BITS
  A nibble is a group of four bits. It can represent numbers from 0 (0000) to 15 (1111). It is the basis for the hexadecimal number system that is used by humans because it is much easier to read. Hexadecimal numbers go from 0 to F (A being 10, B 11, and so on). Hexadecimal numbers are distinguished from the usual decimal number system by prefixing them with 0x (as in 0xFFFF), or by suffixing with h (as in FFFFh).

- BYTE: *bbbbbbbb*
  2 NIBBLES
  8 BITS
  A byte, with a "y", is a group of eight bits. It can represent numbers from 0 (0x00) to 255 (0xFF).

- WORD: *bbbbbbbbbbbbbbbb*
  2 BYTES
  4 NIBBLES
  16 BITS
  A word is a group of sixteen bits. It can represent numbers from 0 (0x0000) to 65,535 (0xFFFF).

- DOUBLEWORD (DWORD): *bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb*
  2 WORDS
  4 BYTES
  8 NIBBLES
  32 BITS
  A doubleword is a group of thirty-two bits. It can represent numbers from 0 (0x00000000) to 4,294,967,295 (0xFFFFFFFF).

#### 3.1.1.2 Registers

Registers are places in the CPU where one can place bytes, words, or doublewords to perform operations on them. There are four different types of registers: general purpose registers, segment & offset registers, index registers, and stack registers. Overviews of these four types of registers follow.

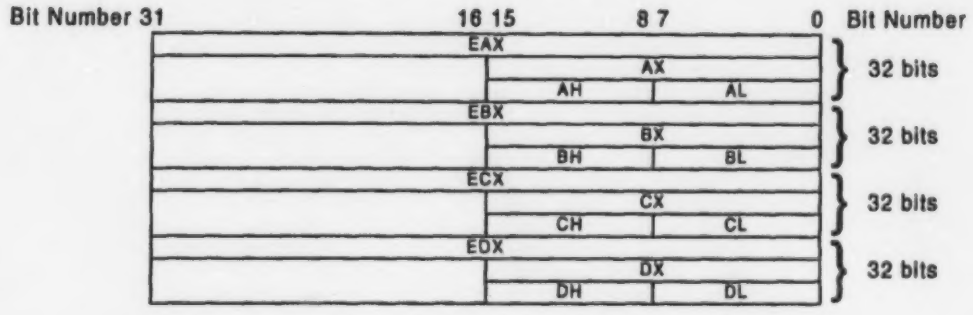


FIGURE 4: The 80x86's general purpose registers

**General Purpose Registers.** As can be seen in Fig. 4, there are four general purpose registers (a through d). They are used to store numbers that are needed to perform calculations (of course, everything in a computer is groups of zeros and ones, so these numbers can be used to represent strings, binary structures, ...). The first 16 bits of `Eax`[1](bits 0 to 15) can be used separately as the 16-bit register `ax`. In turn, `ax` is split in two bytes, the lower one (`al`), and the higher one (`ah`). Using the smaller parts does not change the higher parts. The same goes for `Ebx`, `Ecx`, and `Edx`.

For example, if we want to store the number 24,689, we need 16 bits. So, one can use `ax` `0110000001110001` to store the number. This means that one has `ah` `01100000`, and `al` `01110001`. If one adds one to `al`, it does not change `ah`, but `ax` now represents 24,690[2].

Although they are not split, `Esi`, `Edi`, `Esp`, and `Ebp` (see below) can also be used as general purpose 32-bit registers, but they have more specific uses, as will be seen.

**Index Registers.** Index registers are mainly used for string instructions. There are three index registers, which are sometimes called pointer registers: `ESI` (Source Index), `EDI` (Destination Index), and `EIP` (Instruction Pointer). This last register, `EIP`, is used as an index register, but it cannot be manipulated directly because it stores the address of the next instruction to be executed on the processor.

**Stack Registers.** `Ebp` and `Esp` are stack registers. They are used when dealing with the stack. More details will be given later on when discussing the stack.

[1]The `E` in every register names stands for Extended. The extension was introduced on 386's processors, to pass from a 16-bit architecture to a 32-bit one, while preserving backward compatibility. Also, we use the syntax of Visual C++ and the Win32 APIs for everything. Elsewhere, for instance, one might see `EAX`, or `eax`, instead of `Eax`.

[2]If there is a carry, the Carry Flag (CF) is set. The 1 is not carried to `ah` automatically during operations on `al`.

**Segment & Offset Registers.** Originally, the designer of the 80x86 decided that no one would ever need more than 1 megabyte of memory ($2^{20}$ bits). The processors only had 16-bit registers, so Intel's designers had a problem because it takes 20 bits to represent one megabyte. To solve this problem, they came up with the idea of using 2 registers to store a memory address. One they called the segment register, and the other they called the offset register. What they do to get the address is to multiply the segment register by 16, and add the offset to get a 20-bit number. The notation used to represent an address in a segment S with an offset O is S:O.

This was the only memory model in use before the 386. After the 386, it was still possible to use that form of memory management in what is called the real-mode memory model. However, a much more powerful model exists, the protected-mode memory model. It is the model used by Windows NT and Windows 95/98. It will be discussed in detail in Section 3.3. In this model, segments are called selectors and are used as an index to a descriptor table. The registers are still called segment registers. The segment registers (selectors) are: `SegCs`, `SegDs`, `SegEs`, `SegFs`, `SegGs`, and `SegSS`.

Despite the fact that any register can be used as an offset register (except for segment registers), there are dedicated offset registers: `Ebx`, `Edi`, `Esi`, `Ebp`, `Esp`, `Eip`. As stated previously, this last one cannot be manipulated directly, but it is used as an offset to point to the next instruction to be executed.

### 3.1.1.3 The Stack

As there are only a very limited number of registers, one might wonder how to obtain enough "variables" for one's task. The solution is to use a stack, an area of memory where one can save to and restore values from.

It is called a stack because, just like a stack of plates, it is a pile of 32-bit entities. The last item that is put on the stack is the first one to come off of it. This scheme is called Last On First Out (LOFO), or Last In First Out (LIFO).

In a program, there is a fixed amount of memory reserved for the stack. The stack begins at a relatively high memory address. As one "pushes" new things on the stack, it grows downward to a lower memory address. If too many items are pushed, they go beyond the reserved memory (the stack overflows). Further pushes will write in places where they are not supposed to, thus corrupting memory.

The reverse operation of a "push" is called a "pop". When one pops something, the stack pointer moves up one step. The situation is graphically represented in Fig. 5 (Ref. 10).

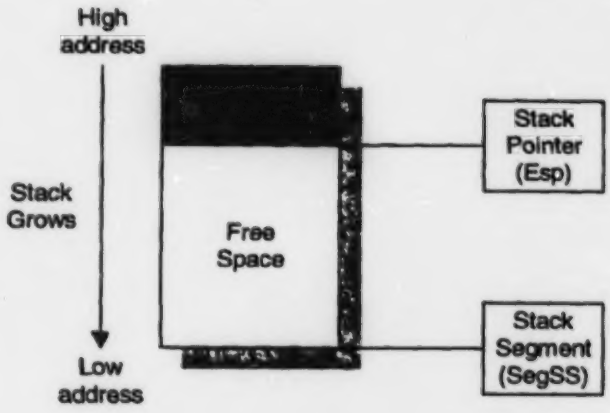


FIGURE 5: The stack

### 3.1.2 Introduction to Assembly Instructions

There are over 150 instructions in 80x86 assembly, — instructions are often called opcodes — most of them with many variations. When debugging, one does not need to know all of them. In fact, only about 20 are frequently used and one will quickly learn to recognise them. This section describes the essential ones. For a description of the complete Intel Processor instruction set, refer to Volume 2 of the Intel manual (Ref. 3).

First, one needs to know how operands work. An operand is a parameter that is given to an instruction. There are three types of operands:

- *Immediate.* A number, or a string, that is known at compile-time and that will always remain the same. Ex: 20, "This program is protected by international laws.", "A", and so on.
- *Register.* Any of the registers that we have seen in the previous subsection.
- *Memory.* A variable that is stored in memory.

This last type is the trickiest of the three in our case since a declared variable looks very different in the source assembly than in a disassembled program. In the source assembly, one declares a variable by giving it a name and the amount of space it takes. Three keywords exist to specify the size: `DB` to declare a byte, `DW` to declare a word, and `DD` to declare a doubleword. For example, one can say `Message DB ''Hello World!$''` to declare `Message` as a pointer to a string — notice the dollar sign as the termination symbol — or `Foo DW ?` to declare `Foo` as an uninitialised variable of 16 bits.

Unfortunately, when an assembly program is compiled, it is optimised. The initialisation data is put in a specific zone and the names of the variables are lost. So, if `push Message` is in the initial program, it will have something like `push 004061CC` in the disassembled program. However, a good disassembler indicate that `004061CC` corresponds to `''Hello World!$''` in the data section of the program.

Now for the essential instructions.

**`Arithmetic.`** Here are the usual arithmetic functions.

**`add.`** Adds the first operand to the second one and stores the answer in the first operand — which of course cannot be an immediate value.

**`sub.`** Subtracts the second operand from the first one and stores the answer in the first operand — which of course cannot be an immediate value.

**`mul.`** Multiplies the one and only operand and implicitly uses `Eax`, `ax`, or `al` as first operand and to store the answer, depending on the size of the numbers.

**`div.`** Divides the number that is placed in AX by the first operand. The result is implicitly stored in AH : AL, DX : AX, or EDX : EAX (pair result:remainder), depending on the size of the numbers.

**`call.`** With this instruction, one calls a "function" — also called a procedure, depending on the context — that will return with the instruction `ret`, after executing its task.

Here again, in the source code, one would call a label name, whereas in the disassembled code, an address will be called; label names are lost.

**`cmp.`** Compares two values. This instruction is usually followed by a conditional jump (see `jumps`). For example:

```
cmp al, 'Y' ; compares value in al with letter Y
je ItsYes   ; if it is equal to Y, jump to ItsYes
```

**`int.`** Produces an interruption that is handled by the system. Typical functions like change video mode, open a file, print a string to the screen, and so on are available through this means[3]. For example:

```
int 21h ; calls a DOS service
int 10h ; calls the video BIOS interrupt
```

[3]In Win32, these interruptions are severely limited and controlled. One usually needs to use the Win32 APIs instead.

TABLE I: Essential conditional jump instructions.

| | |
|---|---|
| `ja` | Jumps if the first number is above the second number. |
| `jae` | Jumps if the first number is above or equal to the second number. |
| `jb` | Jumps if the first number is below the second number. |
| `jbe` | Jumps if the first number is below or equal to the second number. |
| `jc` | Jumps if carry flag is set (same check as jb). |
| `je` | Jumps if the two numbers are equal (`jz`). |
| `jl` | Jumps if the first number is less than the second number. |
| `jna` | Jumps if the first number is not above the second number (`jbe`). |
| `jnae` | Jumps if the first number is not above or the same as the second number (`jb`). |
| `jnb` | Jumps if the first number is not below the second number (`jae`). |
| `jnbe` | Jumps if the first number is not below or equal to the second number (`ja`). |
| `jne` | Jumps if the two numbers are not equal (`jnz`). |
| `jnz` | Jumps if the difference between the two numbers is not zero. |
| `jz` | Jumps if the difference between the two numbers is zero. |

**`jumps`.** The processor recognises a difference between three types of jumps, depending on the operand. There are short jumps that take an 8-bit operand, there are near jumps that take a 16-bit operand, and there are far jumps that take a 32-bit operand.

There is the unconditional jump command, `jmp`, that will jump every time, and there are many conditional jumps, that will jump if a condition is met.

Table I presents most of the common conditional jump instructions. In this table, it can be seen that different mnemonics are used to mean exactly the same thing. They are exactly alike and are provided so that the the most logical one can be used in a given context. However, some instructions that may look the same do not have the exact same implementation. For example, `jl` is triggered when the Sign Flag (SF) is different from the Overflow Flag (OF), while `jb` is triggered when the Carry Flag (CF) is set — more about these flags in 3.2.2.

**`mov`.** Moves a value from one place to another. For example:

```
mov ax, 10      ; moves an immediate value into ax
mov bx, cx      ; moves a value from cx into bx
mov dx, Number ; moves the value of variable Number into dx
```

**`pop/push`.** `push` simply puts the operand on the top of the stack. `pop` does the reverse by taking the top of the stack and putting it into the operand. For example:

```
push cx ; This code effectively switches
push ax ; the values of ax and cx.
pop cx  ; (The instruction xchg does
pop ax  ; that directly.)
```

## 3.2 Intel Processor's Architecture for Debugging

In this section, the extensive debugging facilities that are available directly in the Intel processor are surveyed. For complete details, look in Volume 3 of the Intel manual (Ref. 4). The different parts of the `EFlags` register are also described. The `EFlags` register contains most of the useful (debugging-wise) status flags, control flags, and system flags.

### 3.2.1 Overview of the Debugging Support Facilities

The following debugging facilities exist:

- *Debug exception (*`#DB`*).* Transfers program control to a debugger procedure or task when a debug event occurs.
- *Breakpoint exception (*`#BP`*).* Transfers program control to a debugger procedure or task when an `int 3` instruction is executed.
- *Breakpoint-address registers (*`Dr0` through `Dr3`*).* Directly specify the address of up to four breakpoints.
- *Debug status register (*`Dr6`*).* Reports the conditions that were in effect when a debug or breakpoint exception was generated.
- *Debug control register (*`Dr7`*).* Specifies the forms of memory or I/O access that cause the breakpoint to be generated.
- *Trap Flag (*`TF`*).* When set in the `EFlags`, it generates a debug exception (`#DB`) after executing one instruction.
- *Resume Flag (*`RF`*).* When set in the `EFlags`, it suppresses multiple exceptions to the same instruction (if not set, the processor keeps interrupting on the same instruction).

### 3.2.1.1 Debug Registers

The eight debug registers, as shown in Fig. 6, control the debug operations of the processor. These registers can be used like any other registers when it comes to reading from or writing to them.

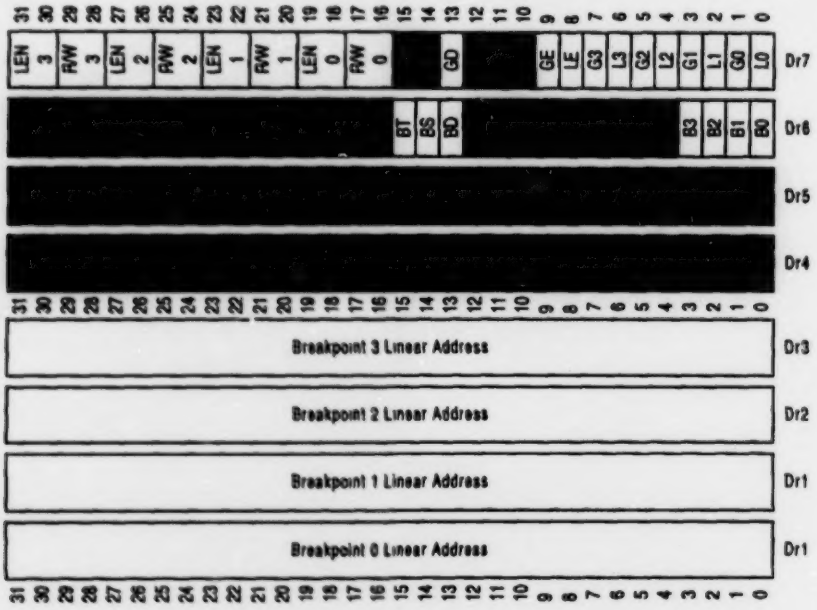


FIGURE 6: The debug registers

The primary function of the debug registers, and the one that concerns us the most, is to set up and monitor 4 different breakpoints, numbered from 0 to 3. Their use will now be explained, along with a description of the major flags (at least, the major ones needed for constructing a debugger).

**Debug Address Registers (`Dr0-Dr3`).** The four debug registers (`Dr0` through `Dr3`) can each hold a 32-bit linear address[4]. When this address is reached, a breakpoint interrupt occurs. Different options can be specified via `Dr7`.

**Debug Registers `Dr4` and `Dr5`.** These debug registers have a special reserved meaning that will not be explained here.

[4]The difference between a linear and a physical address will be described in Section 3.3.

**Debug Status Registers (Dr6).** When an exception occurs, this debug register reports on the debug conditions. Its flags are as follows[5]:

- B0 *through* B3 *(breakpoint condition detected).* When set, they indicate that the associated debug breakpoint condition (Dr0-Dr3) was met.
- BD *(debug register access detected).* If the GD flag is set in Dr7, this flag indicates that the next instruction will access one of the debug registers.
- BS *(single step).* Indicates that the debug exception was triggered by the single-step mode. This is the highest-priority exception, so other flags may be set at the same time.
- BT *(task switch).* Indicates that the debug exception resulted from a task switch.

**Debug Control Registers (Dr7).** This debug register sets breakpoints and their conditions. Its flags are as follows:

- L0 *through* L3 *(local breakpoint enable).* Enable the associated breakpoints (Dr0-Dr3). If $L_n$ is not set, then the processor does not halt at the address in $Dr_n$. These flags are cleared when a task switch occurs to avoid putting breakpoints in an unwanted process.
- G0 *through* G3 *(global breakpoint enable).* Same as the previous except that breakpoints are set for all tasks (they are not cleared on a task switch, which means blocking on this address for all processes).
- LE *and* GE *(local and global exact breakpoint enable).* These flags are not supported in the P6 family processors. They are provided only for backward compatibility.
- GD *(general detect enable)* When set, it generates a debug exception when a process tries to access the debug registers.
- R/W0 *through* R/W3 *(read/write).* Specifies the breakpoint condition for the corresponding breakpoint. This is linked with the use of Dr4 and Dr5 so more details will not be given.
- LEN0 *through* LEN3 *(length).* These flags specify the size of the memory location at the address specified in the corresponding breakpoint address register (Dr0-Dr3). Here again, this is linked with the different parameters of R/W flags. They should be left at 00 when using the debug registers to break on instructions.

[5]It is worth noting that the flags in the Dr6 register are never cleared by the processor. To avoid confusion, the debugger should clear them when necessary.

FIGURE 7: EFlags Register

### 3.2.2 The `EFlags`

Another important thing one needs to know to debug an `80x86` processor is the content of the `EFlags` register. It contains control and system flags that control the flow of execution of programs. The positions of its different flags are shown in Fig. 7 and their descriptions follow. Once again, for complete details, refer to the Intel Manual (Ref. 2) (Volume 1).

- Status Flags
  - `CF` *(Carry Flag).* Set if an arithmetic operation generates a carry or a borrow out; cleared otherwise.
  - `PF` *(Parity Flag).* Set if the least-significant byte of the result contains an even number of bits set to 1; cleared otherwise.
  - `AF` *(Adjust Flag).* Same as `CF` except that it is for binary-coded decimal (BCD) arithmetic.
  - `ZF` *(Zero Flag).* Set if the result is zero; cleared otherwise.
  - `SF` *(Sign Flag).* Set equal to the most significant bit of the result (sign bit). 0 indicates a positive; 1 indicates a negative.
  - `OF` *(Overflow Flag).* Set if the integer result is too large a positive or too small a negative number.
- `DF` *(Direction Flag).* Controls the direction that strings are read in (or written on). 1 indicates auto-decrement; 0 indicates auto-increment.
- Status Flags
  - `IF` *(Interrupt Enable Flag).* Set to respond to maskable interrupts; cleared to inhibit maskable interrupts.

- TF *(Trap Flag).* Set to enable single-step mode for debugging; clear to disable single-step mode[6].
- IOPL *(I/O Privilege Level).* Indicates the I/O privilege level of the currently running program or task.
- NT *(Nested Task).* Set when the current task is linked to the previously executed task; cleared when it is not.
- RF *(Resume Flag).* Controls the processor's response to debug exceptions.
- VM *(Virtual-80x86 Mode).* Set to enable memory in real-mode; cleared to return to protected-mode. More about this in the next section.
- AC *(Alignment Check).* Controls the alignment of memory references.
- VIF *(Virtual Interrupt).* Virtual image of the IF flag. Used in conjunction with the VIP flag.
- VIP *(Virtual Interrupt Pending).* Set to indicate that an interrupt is pending; cleared when no interrupts are pending.
- ID *(Identification).* Giving an explanation of this flag requires knowledge that is out of the scope of the current document.

### 3.3 Protected-Mode Memory

Yariv Kaplan (Refs. 11, 12, 13) has written a very good three-part introduction to protected-mode memory management. The figures in this section are greatly inspired from that document. Once again, all the details of the Intel architecture are available in the Intel Manual, Volume 3 (Ref. 4).

Originally, the Intel processors worked in what is now called "real-mode" memory. Since that time, a much more powerful mode, "protected-mode", was introduced in the 80286 family and the newer processors. Since both Windows NT and Windows 95/98 work in protected-mode, real-mode will not be explained here, not even to illustrate the differences. At any rate, Kaplan explains it well in his article, if the reader is interested in more details. Suffice to say that, for backward compatibility, it is possible to run a task in a virtual-mode that emulates the real-mode, while using the advanced features of the protected-mode.

[6]The flag is cleared after executing one instruction in single-step mode. Thus, it needs to be reset after each execution of one instruction. This is to prevent unwanted and excessive use of single-stepping.

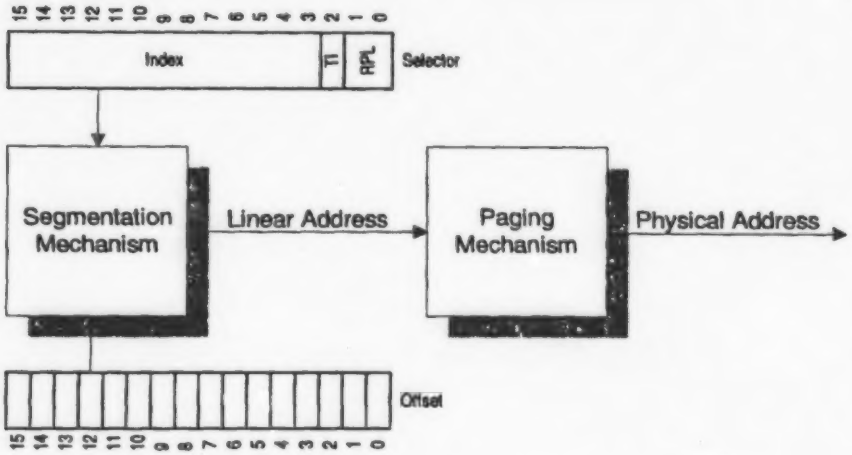
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
Index
TI
RPL
Selector
Segmentation Mechanism
Linear Address
Paging Mechanism
Physical Address
Offset
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

FIGURE 8: Protected-mode address translation process

Since the 80386, the processors have a 32-bit address bus that gives the user a total of 4GB physical address space. This memory is managed by two independent mechanisms, working in tandem: segmentation and paging. The situation is illustrated in Fig. 8.

### 3.3.1 Segmentation

Segmentation allows the operating system to divide memory into logical blocks. These blocks can then be assigned to different programs, for example, and the operating system can then ensure that a program does not write in undesirable places.

As stated in 3.1.1.2, segment registers are used as selectors. A selector contains an Index to a descriptor table, a Table Indicator (`TI`) — which, when set, selects the LDT, and when cleared, selects the GDT, both of which are described below — and a Requestor Privilege Level (`RPL`), that controls the access rights to the memory. See Fig. 8 for the position of these flags.

A descriptor table resides in system memory. These tables are used to perform address translation. Each entry in a table is two doublewords and represents a single segment in memory. An entry contains a pointer to the first byte of the associated segment (`Base Address`), and a 20-bit value (`Segment Limit`) representing the size of the segment. This is shown in Fig. 9.

Figure 9 also shows how the information obtained by the selector, the table entry, and the offset is used to get a linear address.

Finally, two types of descriptor tables are allowed by the processor. The first one is called the Global Descriptor Table (GDT) and should be used to hold entries of system segments. The second

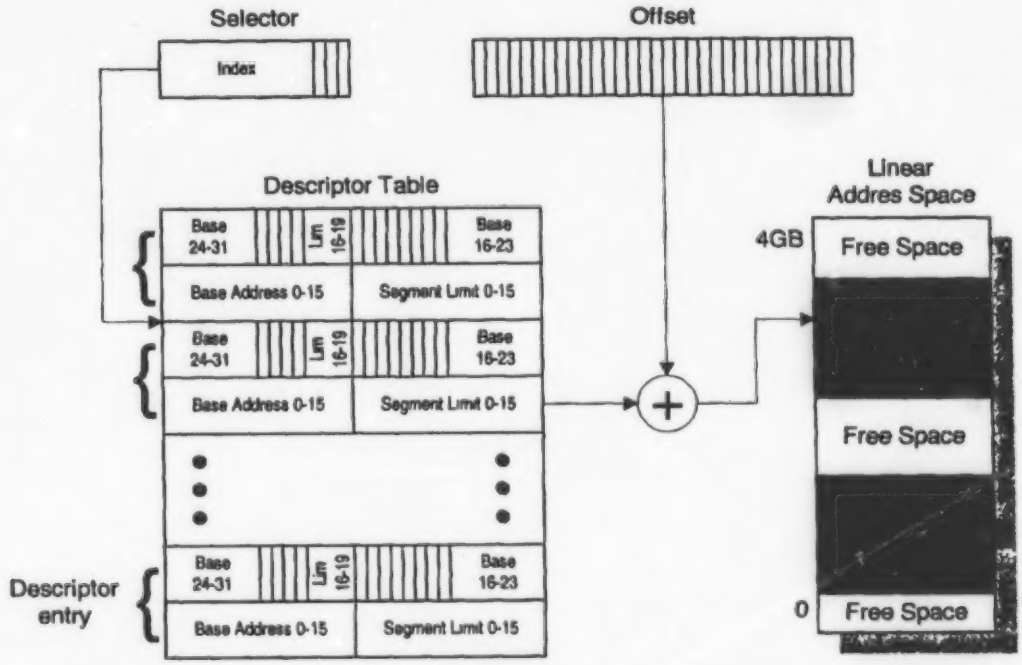


FIGURE 9: Virtual to linear address translation

one is called the Local Descriptor Table (LDT) and contains normal application segments. On the processor, there can be one LDT for each task, but only one GDT. However, Windows NT and Windows 95/98 do not use the GDT and all applications share a single LDT.

### 3.3.2 Paging

As stated earlier, applications in Windows NT (and Windows 95/98) live inside a 4 Gigabyte (GB) linear address space. Applications are not aware of the paging mechanism, and understanding it is not needed to make the debugger work. However, for the sake of completeness, paging will be briefly discussed.

Paging is a mechanism which helps in creating a unique virtual (faked) address space for applications. This allows the operating system to use disk space in the place of memory when a program runs out of it. In Windows NT (and Windows 95/98), this "disk" memory is called virtual memory.

The process works like this. Memory is divided in chunks of 4KB (4MB is possible in Pentiums). When the current application asks for more memory than is available, the system finds the

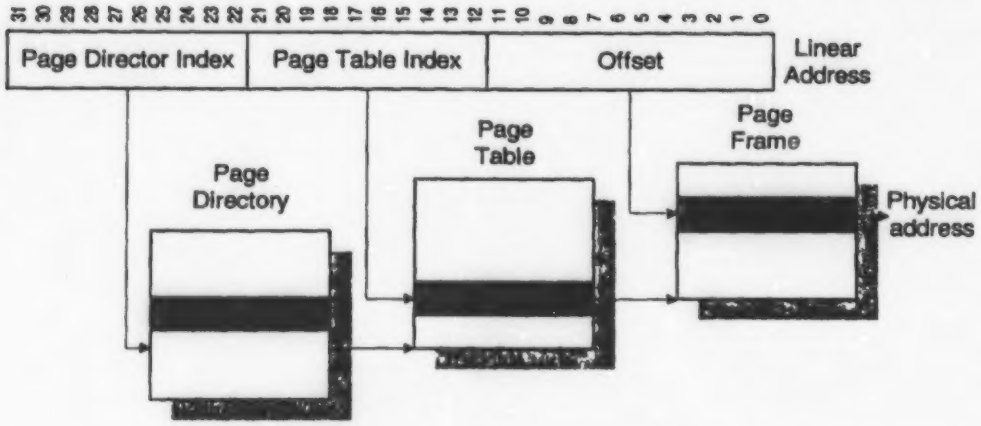


FIGURE 10: Components of a linear address

least recently used (LRU) pages and puts them on disk (page out). Then, it gives this "stolen" memory to the current application. When an application comes alive again and needs to access a block of paged-out memory, an exception occurs that allows the operating system to bring the on-disk memory back into real memory (page in). The application then continues its normal operations, without even being aware that a break occurred.

More precisely, the processor works as follows to find out the physical address, as shown in Fig. 10. It breaks down the linear address into three components. The top 10 bits are used to index the page directory table. An entry of this table indicates the physical address of a page table. The next 10 bits of the linear address are an index to this table. An entry in this table gives the physical address of the beginning of a page. This address plus the last 12 bits of the linear address (offset) gives the final physical address.

### 3.4 Inside Win32's Debugging API

As stated earlier, Win32 provides a lot of debugging facilities; probably everything that one will ever need. Unfortunately, the documentation from Microsoft is obscure, at best, and totally lacking in some areas. With the "inside information" given in the previous sections of this chapter, the reader should be able to better understand the Win32 API. In addition, a lot more details on the relevant parts are presented here than in the Visual C++ documentation (Ref. 1). Also, the colorfully named Iceman (Ref. 5) gives an informal tutorial on the subject.

In this section, a tutorial on how to create a simple debugger in Win32 is presented. To start with, an overview of the debugging functions and their associated structures is presented. Then, a

skeleton of the main loop that any debugger needs to implement is described. Finally, how to set up breakpoints is explained.

Before continuing, here is the definition of a few generic terms.

- A *debugger* is a program that controls another program and can look at the other program's data.
- A *debuggee* is a process that is being debugged by a debugger.

The two following definitions are taken from the Visual C++ manual (Ref. 1).

- A *process* is an application in memory which is running.
- A *thread* is the basic entity to which the operating system allocates CPU time.

A thread can execute any part of the application's code, including a part currently being executed by another thread. All threads of a process share the virtual address space, global variables, and operating system resources of the process.

### 3.4.1 Debugging Functions

Win32 provides two classes of debugging functions. One class is composed of the functions to be used by the debugger; the other is composed of functions to be used by the debuggee. Since, in the case of COTS products, there is no source code access, this last class cannot be used. However, they are interesting if one wants to create toy programs to test one's debugger, so they are described anyway.

Full details about the parameters and return values of these functions are not presented — they are presented in the manual (Ref. 1) — except when they are relevant to further discussion.

#### 3.4.1.1 Debugger Functions

These functions are used by the debugger process to start, control, and get information from the debuggee process. They are given in alphabetical order.

**`ContinueDebugEvent`.** This function allows the debugger to continue executing the debuggee after an exception occurred. One can specify that the debugger handles the exception, or leave it for the system to take care of.

**`CreateProcess`.** This function creates a new process that one can control from the start.

**DebugActiveProcess.** This function attaches one's debugger to an already running application.

**FlushInstructionCache.** This function clears the instruction cache of a process.

**GetThreadContext.** This function allows the debugger to get the context of the debuggee process at a particular point in time. The CONTEXT structure, in essence, contains the processor's registers. It is defined in 3.4.2.1.

**ReadProcessMemory.** This function allows the debugger to read a region of memory in the debuggee's address space.

**SetDebugErrorLevel.** This function allows the debugger to set the minimum error level of debugging events that are of interest.

**SetThreadContext.** The counterpart of `GetThreadContext`, this function allows the debugger to set the contents of the registers for the applications.

**WaitForDebugEvent.** This function is the main component of the debugger. With it, a debugger signals that it is waiting for the next debugging event to occur. The debugger is woken up when an event occurs, handles it, and signals that it is ready for the next event with `ContinueDebugEvent`. More details will be given in Subsection 3.4.3.

**WriteProcessMemory.** This function allows the debugger to change a region of memory in the debuggee's address space.

#### 3.4.1.2 Debuggee Functions

These functions can be used by a debuggee process to communicate with a debugger process. They are given in alphabetical order.

**DebugBreak.** This function causes a breakpoint exception to occur in the debuggee process. If the exception is not handled by a debugger process, it is most likely that the process will end, saying that an exception was not handled. One can know if a debugger is present by using the function `IsDebuggerPresent`.

**FatalExit.** This function causes the debuggee process to exit and give the control back to a debugger process. Here again, it may be dangerous to use this function if it is not known whether a debugger is running or not.

**IsDebuggerPresent.** This function allows the debuggee to know if it is being debugged.

**OutputDebugString.** This function allows the debuggee to send a string to the debugger.

### 3.4.2 Debugging Structures

In the functions outlined above, some common structures are used. Once again, all the details about all of them are not presented since it is not the goal of this document to repeat the on-line help. Only the main ones that are pertinent to our specific need of executing a program instruction by instruction are presented.

#### 3.4.2.1 The CONTEXT Structure

The CONTEXT structure is used to record the context of the program at a given point in time. The context of a program is hardware dependent. In essence, it records the content of the processors' registers. In this case, for the Intel processor, the relevant registers from 3.1.1.2 are set in the CONTEXT structure, as shown in Code Excerpt 1[7]. There is also a completely undocumented FLOATING_SAVE_AREA variable to control floating points, and ContextFlags that are explained below.

One fills in a CONTEXT structure for a given application with the GetThreadContext function; one can also set the variables for the structure of the debuggee to specific values with the SetThreadContext. Probably for performance reasons, one can get/set only parts of the structure by setting the following flags for the ContextFlags:

- CONTEXT_CONTROL. Get/Set Ebp, Eip, SegCs, EFlags, Esp, and SegSs.
- CONTEXT_INTEGER. Get/Set Edi, Esi, EbxCs, Edx, Ecx, and Eax.
- CONTEXT_SEGMENTS. Get/Set SegGs, SegFs, SegEs, and SegDs.
- CONTEXT_FLOATING_POINT. Get/Set FloatSave.
- CONTEXT_DEBUG_REGISTERS. Get/Set Dr0 through Dr7.

[7]For C++ code, we use the same highlighting as Visual C++ — except that there are no colors in this document — **bold** for reserved keywords and *italic* for comments.

**Code Excerpt 1** The CONTEXT structure

```
typedef struct _CONTEXT {
    DWORD ContextFlags;

    DWORD Dr0;
    DWORD Dr1;
    DWORD Dr2;
    DWORD Dr3;
    DWORD Dr6;
    DWORD Dr7;

    FLOATING_SAVE_AREA FloatSave;

    DWORD SegGs;
    DWORD SegFs;
    DWORD SegEs;
    DWORD SegDs;

    DWORD Edi;
    DWORD Esi;
    DWORD Ebx;
    DWORD Edx;
    DWORD Ecx;
    DWORD Eax;

    DWORD Ebp;
    DWORD Eip;
    DWORD SegCs;
    DWORD EFlags;
    DWORD Esp;
    DWORD SegSs;
} CONTEXT;
```

P515758.PDF [Page: 50 of 63]



**Code Excerpt 2** The DEBUG_EVENT structure

```
typedef struct _DEBUG_EVENT {
    DWORD dwDebugEventCode;
    DWORD dwProcessId;
    DWORD dwThreadId;
    union {
        EXCEPTION_DEBUG_INFO Exception;
        CREATE_THREAD_DEBUG_INFO CreateThread;
        CREATE_PROCESS_DEBUG_INFO CreateProcessInfo;
        EXIT_THREAD_DEBUG_INFO ExitThread;
        EXIT_PROCESS_DEBUG_INFO ExitProcess;
        LOAD_DLL_DEBUG_INFO LoadDll;
        UNLOAD_DLL_DEBUG_INFO UnloadDll;
        OUTPUT_DEBUG_STRING_INFO DebugString;
        RIP_INFO RipInfo;
    } u;
} DEBUG_EVENT;
```

- CONTEXT_FULL. This flag has the same effect as setting CONTEXT_CONTROL | CONTEXT_INTEGER | CONTEXT_SEGMENTS.

#### 3.4.2.2 The DEBUG_EVENT Structure

The DEBUG_EVENT structure contains the debug exception information when it is raised and caught by the WaitForDebugEvent function. The definition of the structure is given in Code Excerpt 2.

In this structure, dwProcessId, and dwThreadId identify the process whose thread raised the exception. dwDebugEventCode identifies the type of debugging event that occurred. Following this, the appropriate secondary structure in the union is filled in. The different types of debugging events are the same as the different types of structures in the union of the DEBUG_EVENT; one gets their names by replacing the _INFO part by _EVENT. For example, the structure to get information about a debug exception is EXCEPTION_DEBUG_INFO while the flag to identify this debug exception is EXCEPTION_DEBUG_EVENT.

These information structures are fairly well documented in the Visual C++ manual (Ref. 1), so they are not all repeated here. Only details about the structure named EXCEPTION_DEBUG_INFO are presented, which is of particular interest in debugger construction.

P515758.PDF [Page: 51 of 63]



**Code Excerpt 3** The EXCEPTION_DEBUG_INFO and EXCEPTION_RECORD structures

```
1  typedef struct _EXCEPTION_DEBUG_INFO {
2     EXCEPTION_RECORD ExceptionRecord;
3     DWORD dwFirstChance;
4  } EXCEPTION_DEBUG_INFO;
5
6  typedef struct _EXCEPTION_RECORD {
7     DWORD ExceptionCode;
8     DWORD ExceptionFlags;
9      struct _EXCEPTION_RECORD *ExceptionRecord;
10    PVOID ExceptionAddress;
11    DWORD NumberParameters;
12    DWORD ExceptionInformationEXCEPTION_MAXIMUM_PARAMETERS;
13 } EXCEPTION_RECORD;
```

#### 3.4.2.3 The EXCEPTION_DEBUG_INFO Structure

When a debug exception is identified by the dwDebugEventCode of the DEBUG_EVENT structure, an EXCEPTION_DEBUG_INFO structure is filled in. This structure works in two steps. First, it fills in an EXCEPTION_RECORD. Second, it provides a variable (dwFirstChance) to detect if this is the first or second chance to handle the exception. This second chance mechanism is part of Win32's handling of exceptions. This is illustrated by the definition of the structure in the first four lines of Code Excerpt 3.

This code excerpt also introduces the EXCEPTION_RECORD structure, which is similar to DEBUG_EVENT in its behaviour in that it contains a code to identify what type of exception occurred (ExceptionCode). It contains an exception flag to indicate if it is a continuable exception or not, and the address of the instruction that caused the exception. Also, exception records can be linked together to provide more information on nested exceptions. Furthermore, when the exception is an access violation, the ExceptionInformation array is filled in with NumberParameters parameters.

There are 22 different identification codes. Among the most interesting for our purposes are:

- EXCEPTION_ACCESS_VIOLATION
- EXCEPTION_BREAKPOINT
- EXCEPTION_SINGLE_STEP

Also identified are stack overflows, array bounds exceeded, misaligned data, divide by zero, and so on. It is worth noting that when the program breaks on an address that was specified in a

debug address register (as explained in 3.2.1.1), it is not identified by `EXCEPTION_BREAKPOINT` as we could expect, but rather by `EXCEPTION_SINGLE_STEP`.

### 3.4.3 The Main Debugging Loop

Putting together the information of the previous subsections, it is time to give a sample of the main loop that is executed by a debugger.

A code skeleton for the loop is given in Code Excerpt 4. Here are some details.

In the first five lines, the needed variables are created. Most of them have already been explained in previous subsections. `ProcessInfo` and `StartupInfo` are used by the `CreateProcess` function to set/get information from the debuggee process that will be started.

At line 8, the context flag is set to get/set the control, debug, and segment registers. This value remains in effect until one changes it.

Then, at line 10, the debuggee process is created. If the creation is successful, execution continues; if it is not, this condition is treated it at lines 53-54.

At line 14, the main debugging loop starts. It will end only when the debuggee process terminates, or when it voluntarily relinquishes control. Termination is checked at lines 41 to 43. When the debuggee process terminates, it raises an exception. When this exception is caught, simply raise the flag `endLoop`.

Lines 22 to 37 handle the case where an exception debug event occurs. At line 23, another switch is used to handle the debug exception according to its type. Lines 30 and 31 are worth a special note. As has been seen in Subsection 3.2.2, when the processor is put in single-step mode, it raises an `EXCEPTION_SINGLE_STEP` only for the first exception. To continue single-stepping, the single-step flag (bit 8 of the `EFlags`]) needs to be reset.

The other cases are pretty much self-explanatory.

At line 50, the function `ContinueDebugEvent` is called. It continues the execution of the debuggee until another debugging event occurs, which will be caught by the next loop.

Finally, all debug events and all different types of debug exceptions should be handled to construct a complete debugger.

### 3.4.4 Setting up breakpoints

Now, almost all of the information needed to implement a debugger has been presented. The only thing left is to be able to set breakpoints. To do that, there are three choices:

- Replace an instruction with an `int 3` to cause a breakpoint. After the breakpoint, put the instruction back in its place and reset the instruction pointer (`Eip`) one step behind to

**Code Excerpt 4** The main debugging loop

```
PROCESS_INFORMATION processInfo;
STARTUP_INFORMATION startupInfo;
DEBUG_EVENT debugEvent;
DWORD dwContinueStatus;
CONTEXT appContext
// To get the context for the control, the segments, the debug, and the
// general purpose registers, set the following:
appContext.ContextFlags = CONTEXT_FULL | CONTEXT_DEBUG_REGISTERS;

if( CreateProcess(0, "target.exe", 0, 0, FALSE, DEBUG_ONLY_THIS_PROCESS,
   0, 0, &startupInfo, &processInfo) ) {
bool endLoop;

while( !endLoop ) {
      WaitForDebugEvent(&debugEvent, INFINITE);

      dwContinueStatus = DBG_CONTINUE;
      // To make the context of the debuggee available to all cases:
      GetThreadContext(pi.hThread, &appContext);

      switch(debEvent.dwDebugEventCode) {
      case EXCEPTION_DEBUG_EVENT:
          switch(debEvent.u.Exception.ExceptionRecord.ExceptionCode) {
          case EXCEPTION_BREAKPOINT:
              // Handle a breakpoint exception.
              break;
          case EXCEPTION_SINGLE_STEP:
              // Handle a single-step exception. Here, if we want to continue to
              // single step, we need to do the following.
              appContext.EFlags |= 0x00000100; // Set Trap Flag
              SetThreadContext(pi.hThread, &appContext);
              break;
          case EXCEPTION_ACCESS_VIOLATION:
              // Handle an access violation.
             break;
          }
          break;
      case OUTPUT_DEBUG_STRING_EVENT:
          // Handle an ouput string from the debuggee.
          break;
      case EXIT_PROCESS_DEBUG_EVENT:
         endLoop = TRUE;
          break;
      default:
          // Default behaviour for exception that we do not want to handle.
         dwContinueStatus = DBG_EXCEPTION_NOT_HANDLED;
      }
   }

   ContinueDebugEvent(debugEvent.dwProcessId, debugEvent.dwThreadId,
                      dwContinueStatus);
} else {
   // Problem creating the process.
}
```

**Code Excerpt 5** Putting in a breakpoint (`int 3`)

```
1 DWORD dwAddress = 0040614D;
2 DWORD dwbreakPointOpcode = 0xCC;
3 int numberOfBytesToWrite = 1;
4 WriteProcessMemory(processInfo.hProcess, (LPVOID)dwAddress,
5                     &breakpointOpcode, numberOfBytesToWrite, NULL);
```

reexecute at the same address.

- Put the processor in single-step mode to break on every instruction.
- Use the four debug registers to set up breakpoint addresses.

The last solution is perfect and simple if one needs only up to four breakpoints. Since that is not often the case, another solution should probably be used.

The second solution is cumbersome for many reasons. Firstly, breaking on every instruction will slow down the debuggee process unreasonably. Secondly, single-stepping a process will stop on every instruction, in whatever module it is in. For instance, this would occur every time it enters a Dynamic Link Library (DLL). If one works on the assumption that Windows NT's components are trustworthy, they do not need to be examined. Thus, this solution is only practical for small and controlled code segments.

The first solution is probably the most general, and therefore the most practical one. So, how does one put `int 3` in the place of an instruction?

To achieve this, one uses the `WriteProcessMemory` function. The only "problem" is that one needs to know what an `int 3` instruction looks like in assembly. Looking it up in Volume 2 of the Intel manual (Ref. 3), it says `CC`. All of the needed pieces are now in place. Suppose the address of the instruction is `0x0040614D`, the function call should look something like what is shown in Code Excerpt 5.

Of course, the instruction that is there in the first place should be recorded somewhere (via `ReadProcessMemory`), in order to be able to put it back after the interruption.

## 4.0 CONCLUSION

Detecting a malicious code is not trivial. In order to understand the various aspects of this problem on an Intel machine running Windows NT, it is clear that the following criteria are necessary:

- Knowledge of the inner workings of NT.
- Knowledge of the security features of NT.
- Knowledge of the instructions available on Intel and its architecture.
- Knowledge of the Win32 subsystem and its debugging features.

Even if these techniques do not prove to be useful in the actual detection of malicious code, they are still necessary to provide the needed information gathering tools. Since most malicious codes exploit very low-level calls and are often done in assembly for optimisation, it is clear that the information presented in this report is necessary.

Of course, these techniques could also prove to be useful at detecting malicious code. It would be a relatively simple task to build an advanced debugger that would stop at critical API calls. It could then keep track of the variables used by this API and make sure that they do not cause harm.

For example, say that one wants to make sure that a program does not "expire" after a given time (a time bomb). An advanced debugger could keep a list of all APIs that can give access to the system time. Then, it would continue execution, looking at the relevant variables. If the flow of control is influenced by today's date, it could inform the user, stop the execution, or force the execution to continue on the other branch. This requires very intimate and extensive knowledge of time bombs. The information presented in this report can certainly be used to gather this information with a specialised, in-house or commercial debugger.

After an initial phase of information gathering, it remains to be seen if an advanced debugger for the detection of malicious code would be a practical solution both in terms of speed and usefulness compared with simpler techniques.

In recent years, malicious code has risen to become a major concern for many companies. This is due, in most part, to the proliferation of mobile code via the Internet (e.g. Java, JavaScript, ActiveX, CGI script). A consortium of companies has even been created to address this problem. Managed by the International Computer Security Association (ICSA), the Malicious Mobile Code Consortium (MMCC) has the following mission:

> "... to enhance computer security from malicious executable content through consumer education, by developing product certification standards, and creating public awareness of active content security products." (Ref. 14).

In this spirit, many tools exist that try to protect a computer or a network from such code. Unfortunately, almost all of these tools address only the problem of *mobile* code; even then, they are quite weak as security goes. Furthermore, there is no provision for COTS software that is installed from a CD. The goal of the MaliCOTS project is precisely to tackle these shortcomings.

## 5.0 REFERENCES

1. Microsoft Corporation. *Microsoft Visual C++ On-line Documentation*, 1997.

2. Intel Corporation, Mt. Prospect IL, USA. *Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture*, 1997. Order Number 243190, `http://developer.intel.com/design/pentiumII/manuals/243190.htm`.

3. Intel Corporation, Mt. Prospect IL, USA. *Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference Manual*, 1997. Order Number 243191, `http://developer.intel.com/design/pentiumII/manuals/243191.htm`.

4. Intel Corporation, Mt. Prospect IL, USA. *Intel Architecture Software Developer's Manual, Volume 3: System Programming Guide*, 1997. Order Number 243192, `http://developer.intel.com/design/pentiumII/manuals/243192.htm`.

5. Iceman. Win32 - Inside Debug API. `http://fravia.org/iceman1.htm`, March 1998.

6. Art Baker. *The Windows NT™ Device Driver Book: A Guide for Programmers*. Prentice Hall, Upper Saddle River, NJ, USA, first edition, 1997.

7. Stephen A. Sutton. *Windows NT Security Guide*. Addison-Wesley Developpers Press, first edition, December 1996.

8. National Security Agency. Microsoft Corporation Windows NT Workstation and Windows NT Server Version 3.5 with Service Pack 3. `http://www.radium.ncsc.mil/tpep/epl/entries/CSC-EPL-95-003.html`, July 1995.

9. Robert Malmgren. NT Security - Frequently Asked Questions. `http://www.it.kth.se/~rom/ntsec.html`, 1997. Version 0.41.

10. Gavin Estey. Gavin's Guide to 80x86 Assembly. `http://carp.rutgers.edu/cs/assembly/asm1.htm`, 1996.

11. Yariv Kaplan. Introduction to Protected Mode. `http://www.internals.com/articles/protmode/introduction.htm`, 1997.

12. Yariv Kaplan. Protected-Mode Memory Management. `http://www.internals.com/articles/protmode/protmode.htm`, 1997.

13. Yariv Kaplan. Real-Mode Memory Management. `http://www.internals.com/articles/protmode/realmode.htm`, 1997.

14. International Computer Security Association (ICSA). Malicious Mobile Code Consortium (MMCC). http://www.icsa.net/services/consortia/mmc/, 1998.

## INDEX

access rights, 6, 11–13, 29
access violation, 37
ACL, 10–12, 15, 16
address
    linear, 25, 29–31
    physical, 25, 29, 31
    virtual, 6, 7, 30, 32
API, 8, 9
    debugging, 31–40
audit, 7, 14–16

COTS, 1, 32

debuggee, 32
    function, 33–34
debugger, 17, 30–32
    function, 32–33
    main loop, 38
    setting a breakpoint, 38–40
    single-step, 26, 28, 38, 40
debugging structure, 34–38
descriptor table, 20, 29, 30
DLL, 7, 40
dynamic analysis, 17

EFlags, 24, 27–28, 34, 38

Intel, 1, 17, 21, 24–28, 34, 40, 41

malicious, 1, 3, 13
malicious code, 1, 3, 16
memory
    heap, 7
    paging, 7, 29–31
    protected-mode, 20, 28–29
    real-mode, 20, 28
    segment, 6, 29, 30
    segmentation, 29–30
    stack, 7, 17, 20–21, 23, 37
    virtual, 7, 30
    virtual-mode, 28
MS-DOS, 8, 9

register, 4, 17–21, 24, 27, 33, 34
    debug, 24–26, 34, 38, 40
    general purpose, 18–19
    index, 18–19
    offset, 18, 20
    segment, 18, 20, 29, 38
    selector, 20, 29
    stack, 18–19

sniffer, 16
specification, 12

Table Indicator, 29
trapdoor, 1

virtual DOS machine, 8
virus, 16

Win32, v, 2, 8–9, 17–41
Windows 95/98, 8, 9, 20, 28, 30
Windows NT, 1–17, 20, 28, 30, 40, 41
    domain, 10–11, 13, 15
    NTFS, 9–11

**UNCLASSIFIED**

**INTERNAL DISTRIBUTION**

**DREV - TR - 1999 - 145**

1 - Director General

1 - Deputy Director General

6 - Document Library

1 - Mr. M. Salois (author)

1 - Dr. G. Vézina

1 - Mr. D. Demers

1 - Mr. R. Charpentier

1 - Mr. G. Thibault

1 - Dr. É. Bossé

1 - Mr. D. Gouin

1 - Mr É. Dorion

**UNCLASSIFIED**

**EXTERNAL DISTRIBUTION**

**DREV - TR - 1999 - 145**

1 - DRDKIM

1 - DRDKIM (unbound copy)

1 - DRDC
Attn: Mr. B. Eatock

4 - DREO
Attn: Mr. V. Taylor

1 - DDCEI Attn: Mr. B. Allan, DDCEI 3-5

3 - MARLANTHQ N21

1 - DSACCIS

1 - DSAA

1 - École de l'Électronique et des communications des Forces Canadiennes (EEFCFC)

1 - Direction - Ingénierie et Intégration de Réseaux Informatisés (DIIRI)

1 - Communication Security Establishment
Attn: Mr L. Bélanger

1 - Network Vulnerability Team
Attn: Capt L. Dandurand

UNCLASSIFIED
SECURITY CLASSIFICATION OF FORM
(Highest Classification of Title, Abstract, Keywords)

**DOCUMENT CONTROL DATA**

| | |
|---|---|
| 1. ORIGINATOR (name and address)<br>DREV<br>2459 Pie-XI Blvd North<br>Val-Bélair QC G3J 1X5 | 2 SECURITY CLASSIFICATION<br>(Including special warning terms if applicable)<br>UNCLASSIFIED |

3. TITLE (Its classification should be indicated by the appropriate abbreviation (S, C, R or U)
DEBUGGING AND ANALYSING WINDOWS NT PROGRAMS AT THE BINARY LEVEL

4. AUTHORS (Last name, first name, middle initial If military, show rank, e.g Doe, Maj John E )
Salois, Martin

| | | |
|---|---|---|
| 5. DATE OF PUBLICATION (month and year)<br>April 2000 | 6a. NO OF PAGES<br>45 (+18=63) | 6b .NO OF REFERENCES<br>14 |

7. DESCRIPTIVE NOTES (the category of the document, e g. technical report, technical note or memorandum. Give the inclusive dates when a specific reporting period is covered )
Technical Report

8. SPONSORING ACTIVITY (name and address)
DREV
2459 Pie-XI Blvd North
Val-Bélair QC G3J 1X5

| | |
|---|---|
| 9a. PROJECT OR GRANT NO. (Please specify whether project or grant)<br>5BF19 | 9b CONTRACT NO.<br>N/A |
| 10a. ORIGINATOR'S DOCUMENT NUMBER<br>DREV-TR-1999-145 | 10b. OTHER DOCUMENT NOS<br>N/A |

11. DOCUMENT AVAILABILITY (any limitations on further dissemination of the document, other than those imposed by security classification)

- ☒ Unlimited distribution
- ☐ Restricted to contractors in approved countries (specify)
- ☐ Restricted to Canadian contractors (with need-to-know)
- ☐ Restricted to Government (with need-to-know)
- ☐ Restricted to Defense departments
- ☐ Others

12 DOCUMENT ANNOUNCEMENT (any limitation to the bibliographic announcement of this document. This will normally correspond to the Document Availability (11). However, where further distribution (beyond the audience specified in 11) is possible, a wider announcement audience may be selected )

UNCLASSIFIED
SECURITY CLASSIFICATION OF FORM
(Highest Classification of Title, Abstract, Keywords)

UNCLASSIFIED
SECURITY CLASSIFICATION OF FORM
(Highest Classification of Title, Abstract, Keywords)

13. ABSTRACT (a brief and factual summary of the document. It may also appear elsewhere in the body of the document itself. It is highly desirable that the abstract of classified documents be unclassified. Each paragraph of the abstract shall begin with an indication of the security classification of the information in the paragraph (unless the document itself is unclassified) represented as (S), (C), (R), or (U). It is not necessary to include here abstracts in both official languages unless the text is bilingual)

In recent years, civilian and military organisations have been using Commercial-Off-The-Shelf (COTS) software more and more instead of developing their own software. Even though this approach offers many advantages in terms of development delays and costs, it causes great concern when it comes to security assurance since it is much more difficult to test software when the source code is not available, as is the case for most COTS software.

In 1997, a project called MalCOTS was initiated to investigate ways of detecting and eliminating the threat caused by malicious codes that could be present in COTS programs. This report represents a first step toward overcoming the technical difficulties associated with a dynamic approach in which one directly monitors the execution of the program.

To monitor a program's execution at the binary level, basic knowledge of assembly language programming is required. It is also preferable to have a good understanding of the architecture of the operating system in use — in this case Windows NT running on an Intel machine. Finally, to be able to control a program's execution and look at its private data, it is necessary to know how to implement a debugger.

This report presents information that was gathered from numerous heterogeneous sources and experimentation, thus allowing a programmer to implement and use a simple debugger that works at the binary level in Windows NT.

14. KEYWORDS, DESCRIPTORS or IDENTIFIERS (technically meaningful terms or short phrases that characterize a document and could be helpful in cataloguing the document. They should be selected so that no security classification is required. Identifiers, such as equipment model designation, trade name, military project code name, geographic location may also be included. If possible keywords should be selected from a published thesaurus, e.g. Thesaurus of Engineering and Scientific Terms (TEST) and that thesaurus-identified. If it is not possible to select indexing terms which are Unclassified, the classification of each should be indicated as with the title.)

Windows NT architecture
Windows NT programming
Windows NT debugger
Windows NT Security
Intel assembly language programming
Dynamic Analysis
Malicious Code

#515758
CA011150

UNCLASSIFIED
SECURITY CLASSIFICATION OF FORM
(Highest Classification of Title, Abstract, Keywords)

National Defence / Défense nationale

Security Classification of Form
UNCLASSIFIED

CA011150

## DOCUMENT REVIEW PANEL PUBLICATION RECORD

*INSTRUCTIONS:*

1. *This form is to be completed for scientific or technical publications produced by or for DND.*
2. *Send one copy of the completed form to DRDB/DRDCIM, together with the Distribution List, DRDCIM copies and those for distribution (if any)*

### A. BIBLIOGRAPHICAL DATA

| | | | |
|---|---|---|---|
| 1. | ORIGINATOR (organization preparing document)<br>DREV | 2. CONTRACT and/or PROJECT NO.(s) (please specify)<br>TNS 5BF19, MaliCOTS | |
| 3. | SPONSOR (DND Project Officer or Directorate)<br>DREV | 4. PUBLICATION SERIES and NO.<br>DREV TR 1999-145 | |
| 5 | TITLE<br>Debugging and Analysing Windows NT Programs at the Binary Level | 6 TITLE CLASSIFICATION<br>UNCLASSIFIED | |
| 7. | PERSONAL AUTHOR(s)<br>Salois, Martin | 8. DATE OF PUBLICATION<br>December 1999 | |
| 9. | FROM (Name of Establishment)<br>DREV | | |

### B. SECURITY CLASSIFICATION / LIMITATION INFORMATION

10. May document be given UNLIMITED distribution ?
(i.e. is the document unclassified and suitable for release to the general public ?
Classified or designated documents, or those with protectable intellectual property or patents pending may not be given UNLIMITED distribution).

Yes ☒ No ☐

*If document is UNLIMITED, skip to ITEM 18. (P. 2)*

11. **Overall SECURITY classification of document:**
(based on A-SJ-100-001/AS-000 Chap. 6, Annex A)

☐ UNCLASSIFIED
☐ CONFIDENTIAL
☐ SECRET

**DESIGNATION**
or (Chap. 6, Annex B)

☐ PROTECTED A
☐ PROTECTED B
☐ PROTECTED C

*If document is not CLASSIFIED or DESIGNATED (but has LIMITED distribution), skip to ITEM 14 (P. 2)*

12. **OFFICIAL WARNING TERM** (e.g. Canadian / US Eyes Only)
(based on A-SJ-100-001/AS-000 Chap. 6, Paras. 6.68-6.71)

Reasons for classification or warning term :

13. **DETAILS OF CLASSIFIED INFORMATION**

| COUNTRY OF ORIGIN | HIGHEST LEVEL (S, C, DR, PC, PB, PA) | PAGES ON WHICH CLASSIFIED OR DESIGNATED INFORMATION IS CONTAINED | | | |
|---|---|---|---|---|---|
| | | TEXT | TABLES | FIGURES | CLASSIFIED TITLES CITED |
| Canada | | | | | |
| U.S. | | | | | |
| U.K | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |

List classified documents referenced (if any) on additional sheets

Security Classification of Form
UNCLASSIFIED

Security Classification of Form
UNCLASSIFIED

## C. SPECIAL RESTRICTIONS

14. Does the INTELLECTUAL PROPERTY (IP) in the document require an additional «RESTRICTIVE LEGEND» to be applied by DRDCIM ?

☐ No ☐ Yes Documents containing information proprietary to the Originator or to DND, which has not been protected by patent or patent application, should receive the protection of the Restrictive Legend which limits their future release.

**15. DOCUMENT AVAILABILITY**

To which types of ORGANIZATION may the document be released without further reference to the Document Review Panel (DRP) ?

| *Please indicate all categories that are appropriate :* | "X" |
|---|---|
| (a) Contractors in approved countries (see para. 16) | ☐ |
| (b) Canadian contractors (with need-to-know) | ☐ |
| (c) Governments (with need-to-know) | ☐ |
| (d) Defence departments | ☐ |
| (e) Other (specify) | |

**16. FOREIGN DISTRIBUTION**

To which COUNTRIES or INTERNATIONAL BODIES may the document be released without further reference to the Document Review Panel (DRP)?

| *Please indicate all categories that are appropriate :* | "X" |
|---|---|
| (a) None | ☐ |
| (b) NATO countries | ☐ |
| (c) NATO agencies : HQ, SACLANT, STC, etc. | ☐ |
| (d) TTCP countries | ☐ |
| (e) The following countries / bodies (specify) | |

**17. DOCUMENT ANNOUNCEMENT**

To whom may this document be announced ?
«Announcement» involves references to the document appearing in DRDCIM Alert (current awareness), CD-ROM, and bibliographies. In most cases, it is to the Originator's advantage to announce the document to as wide an audience as possible.

**Within Canada :**

(a) ☐ All (b) ☐ Contractors with need-to-know (c) ☐ Governments with need-to-know (d) ☐ DND (e) ☐ DRDB only (f) ☐ *DRDCIM only

* Please provide justification if «DRDCIM only» is selected - this would indicate that, although the document is in the DRDCIM collection, its availability would not be announced outside the Directorate.

**Outside Canada:**

(g) ☐ Yes (h) ☐ No

DRDCIM is not able to announce to selective foreign countries, nor is able to guarantee what further announcement allied defence information services give to documents or bibliographical references provided it to the establishments ; however, distribution statements are respected, and requested distribution outside those guidelines is verified with the originator through DRDCIM.

## D. AUTHORIZATION

**18. Record the meeting number and date of action of Section Head or Establishment / HQ Review Panel.**

595 (DRP Meeting No.) 2 dicembre 1999 (Date)

**Approved by**

17 oct 00
(Section Head)

Enclosures:
Distribution List, etc. ( sheet(s))
..... Copies for DRDCIM and distribution

Security Classification of Form
UNCLASSIFIED